Between Small Complex and Large Simplistic Network Simulators

Marat Zhanikeev
Waseda University
1-21-1 Nishi-Waseda, Shinjuku-ku, Tokyo, 169-0051 Japan
maratishe@aoni.waseda.jp

ABSTRACT

Presently, discrete-even network simulation market is split among a number of popular simulators, such as NS-x, OMNET++, Opnet, and maybe a few others. These tools are very common in research that targets small to medium size networks. Whenever one needs to simulate a large network of many tens of thousands nodes, the ability to use traditional network simulators is questionable. Since performance, and specifically, execution speed, is the number one priority in traditional network simulation, simulation runs are normally implemented as solid executables running in a memory space, to which there is a physical limit. This paper proposes a different paradigm of simulation that specifically aims at the ability to simulate large networks while supporting models in nodes with any level of complexity.

Categories and Subject Descriptors

I.6.8 [**Types of Simulation**]: Discrete event, Distributed; I.6.5 [**Model Development**]: Modeling methodologies

General Terms

Network simulation, large topology

Keywords

Computer networks, simulation, modeling, traffic, scalability, discrete event simulation

1. INTRODUCTION

Many discrete-even simulators exist today in the area of network analysis. Among them, *OMNET++* [1] is probably the most actively developed since its source code has been released to public. *OMNET++* also offers GUI environment for both topology creation and analysis of results that makes it more attractive than *ns2*, which is text-based in both these areas. In commercial domain there is *OPNET*, which also offers a strong GUI support at three major levels of modeling: node, module, and process. These three simulators arguably are the top three simulators in the area of network analysis. Also, since *UML* (Universal Modeling Language) is a well established standard, there are many tools that are able to generate source code structures based on UML descriptions. This technique has also been used in simulation [2] [3] [4].

On the part of the performance of simulation runs, when simulations include tens of thousands of objects, single-executable kind of simulation may simply run out of memory in the middle or even at the beginning of simulation. In fact, some reports have been filed about the physical limit on the number of nodes that can be used in *OMNET++*, which is said to be around several dozen thousand nodes. This number should vary depending on the contents of each particular node.

To solve the scalability problem this paper proposes dynamic loading and unloading of models. This does however affect the performance since it takes time to load and unload models, but offers the advantage in form of virtually unlimited number of nodes that can be used in simulation. While simulation is being executed, the volume of memory occupied by all active models is being monitored at all times, and when a threshold defined at simulation start is surpassed, some models are dumped to the hard disk and are loaded only when an event is scheduled to arrive at an unloaded model.

As for the practical uses of the proposed simulator, it is the only choice for a number of simulation-based research topics. Backbone-level multiple source traffic generation, grid, mesh, and p2p network research, and NGN-related research are only few areas which use extremely large topologies for simulation analysis. The author, in particular, uses the proposed simulator for development and verification of active probing techniques.

2. SIMULATOR DESIGN

Each model in the proposed simulator is defined as one *Model* system-global class and a number of states and transitions, where *Model* class is standard for all models used in simulation. This section discusses other design solutions that support dynamic loading of models proposed in this paper.

2.1 State Machines

Let us consider a simple state machine - a common case with one blocking state normally called `IDLE` and a non-blocking state called `process` that is used to process an interrupt and immediately return back to `IDLE` where the model

`Conference name:` SIMUTools 2008, March 3-7, 2008, Marseille, France
`Copyright number` (LaTex): TBA

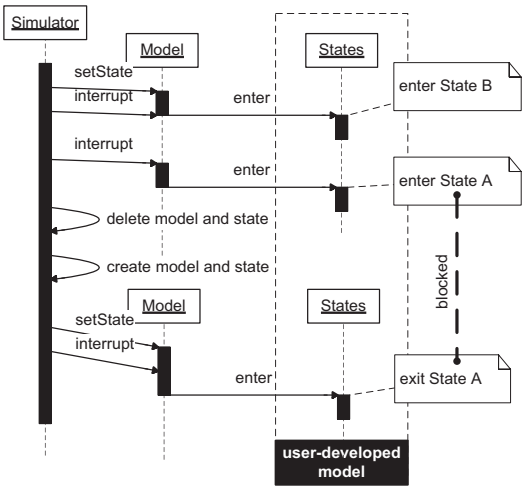

Figure 1: Role of Model class in changing states in models developed by users.

would wait for the next interrupt. Most existing simulators discriminate between non-blocking and blocking states.

Many simulators implement blocking behavior by placing two pieces of code in each state. For example, *OPNET* refers to these pieces as `ENTER` and `EXIT` instructions. Whatever the term, the purpose of enter and exit instructions is common for all simulators that use this concept.

Enter instructions of a blocking state are important since they perform all necessary calculations immediately before the state blocks. For example, if two models were exchanging data, enter instructions of a blocking state would contain the code that schedules an event to arrive at the other model. The state itself would block after having scheduled the interrupt.

Exit instructions are executed when the state receives another interrupt while being blocking. The purpose of exit instructions is dual. Of course, they can be used to process the contents of the interrupt and even send data to some other model within the simulation. However, more common use for exit instructions is preparation of the transition from the blocking state to some other state within the same model. Depending on the type of the interrupt arriving at the blocking state, based on the contents of the interrupt and values of current state variables exit instructions of blocking state can prepare the transition to another state within the same state machine.

The simulator proposed in this paper also supports enter and exit instructions both for blocking and non-blocking states. When the state is declared non-blocking its enter and exit instructions are executing continuously without blocking, which means that exit instructions can be placed together with enter instructions and vice versa. The subject of transitions among states will be discussed later in this paper.

2.2 Model Class and Dynamic Loading

The proposed simulator defines the *Model* class, which is used by all user-developed models. The role of the *Model* class is described in Fig.1. For simplicity reasons all states within user-developed model are placed into the combined execution thread called *States*.

Fig.1 represents two separate concepts. One concept is the relay of interrupts issued by the simulator over to the states within user-defined model. Another concept is dynamic loading of models.

The relay of interrupts is fairly straightforward. Since *Model* class is the same in all user-defined models, the method of relaying interrupts is the same for all interrupts and all models. As will be shown later, this is implemented in form of *interrupt* method defined in *Model* class, which takes *Event* object as the only argument. Given that each *Model* class instance stores the current state of user-defined model, it knows exactly which state to call at all times.

Dynamic loading also fits within the design depicted in Fig.1. To allow dynamic loading, each *Model* class instance has to be abstracted from user-defined models. This is done by defining *setState()* method in *Model* class. When a model is unloaded, its current state is destroyed along with its *Model* class instance, thus releasing the memory used by the model. To restore it, the opposite action is performed, i.e. the current state of the model is recreated, then a new instance of *Model* class is created and its method *setState()* is called to place it exactly in the state at which the model was unloaded from the memory.

It should be noted that since all state variables of a model are stored separately from model states and *Model* class itself, each model can be unloaded without the fear of losing data. The details about loading and unloading models and model state variables will be given later in this paper.

It is also clear that each loaded and unloaded model is always in its blocking state, i.e. having executing enter instructions of a blocking state and waiting for the next interrupt. Naturally, when the model is loaded next time, it is also placed into the blocking state, and when the next interrupt arrives, exit instructions are executed. Class and method structures will be discusses later in this paper.

2.3 Context Switching

It was already mentioned that state variables owned by each user-defined model are stored separately from both user-developed model states and the *Model* class. The *DataRepository* class in Fig.2 is used to store all state variables for all currently active models in simulation.

Fig.2 also performs the loading/unloading sequence. Regardless of whether a model has just been loaded or whether it remained in the memory since the last interrupt, at each interrupt the *Model* class checks whether the context is present in it. If the model has just been loaded, the context in the *Model* class will be set to `NULL`, which is when the *Model* class restores its context by calling a method in *DataRepository* class.

The optimization of context switching is not considered in this paper and is left for future work on the proposed simulator. It is obvious that some improvements can be introduced to performance, for example, by allowing the *Model* class to skip requesting *DataRepository* class for context in case the user-defined model has not registered any state variables. Since model contexts are stored in a list, context searches cause major performance degradation.

In the proposed simulation, even when the model has no state variables registered, an empty context slot is created in the list within *DataRepository* class instance.

2.4 Event Scheduling

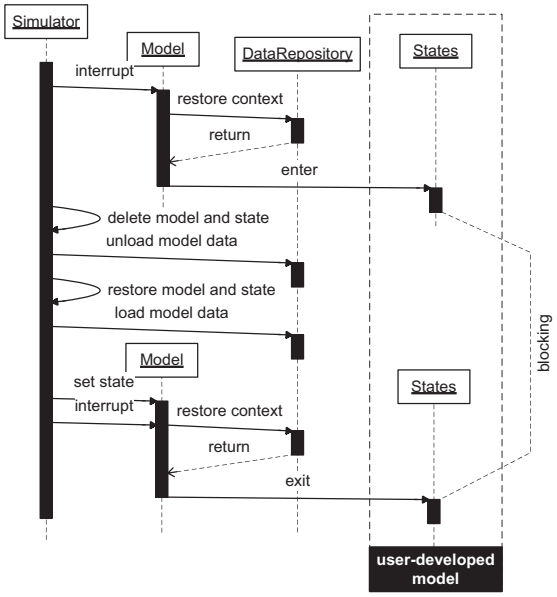

Figure 2: Context switching including loads and unloads of models.

Since unique ID numbers are attributed to models at simulation initiation, a model might not know the ID of the model to which an event is to be scheduled. For this purpose, ID can be obtained using a globally accessible method, which takes the name of the node for an argument and returns its ID which can be used to schedule an interrupt.

The *ModelRepository* class is created specifically for this purpose and contains a list of all models created by the simulator both at the initiation or dynamically during simulation. Each slot in this list contains model ID, name of the model, and name of the model instance. Names of the model and its instance are different since one model can have many instances. In this case all instances will share the model name, but instance names will be different. In fact, instance names are unique throughout simulation regardless of what model was used to instantiate them.

3. IMPLEMENTATION DETAILS

The proposed simulator is written in *C++*. All objects in it are defined as classes, where each class may have multiple instances within the simulation. For example, *Model* class is used in each user-defined model. This section contains details on implementation of the proposed simulator.

3.1 Class Mapping of State Machines

Fig.3 contains *UML Static Structure* diagrams of the three basic classes used in each model. As was mentioned before, each model consists of user-defined states and transitions and the *Model* class, defined in the simulator code.

As was also mentioned earlier, each *Model* class instance stores the current state of the state machine it belongs to. This is represented by class-wide *curstate* variable, which can be set using *setCurState(State state)* method defined in the *Model* class. Naturally, in *C++*, a pointer to *State* class instance would be passed, but this much detail is excessive when discussing the structure of classes.

Another method defined in the *Model* class is *interrupt(*

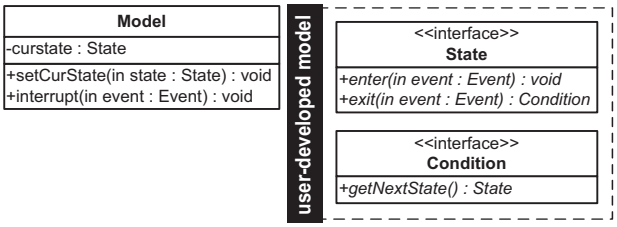

Figure 3: The Model class and two basic interfaces that have to be used in order to develop individual models.

Event event) method, which is used to schedule interrupts to the state machine classes by this *Model* class instance. Each interrupt finally is translated into *enter()* and *exit()* calls on individual states within user-defined state machine.

State and *Condition* classes in Fig.3 are declared as interface classes. In *Java* programming language this would correspond to literally declaring a class as *interface*, but since *C++* has no concept of interfaces, another technique is used. The *State* and *Condition* classes are declared as classes, but their methods are declared using the *virtual* keywords. The use of this keyword allows other classes to extend the class by using the keyword *public* in the class declaration and re-declaring and re-implementing the interface methods in the extension class. Using this technique, one can reach the same result as using the *interface* keyword directly in *Java*.

The *State* and *Condition* interface classes are declared in the header file of the simulator, but user-defined models can be developed elsewhere and do not have to merge their header files with those of the simulator. They, however, have to include *simulator.h* to be able to operate with the interface classes.

Given that some models can be fairly complex and can have many states and multiple conditions facilitating transits among states, user-defined models may define many classes for each model.

This design, however, allows for automatic generation of source code, given that all states and all conditions use the same interface classes. The proposed simulator does not offer a GUI environment yet, but in future work GUI will be implemented and will generate classes for each model automatically.

3.2 Loading and Unloading Models

As was mentioned earlier, a sound programming practice dictates that models are to be developed outside of the simulator source tree. The only connection between user-defined models and the simulator is the *simulator.h* header file, that contains definitions of *Model*, *State* and *Condition* classes, which should be used in user-developed code. The code of the model itself should be placed outside of the tree.

The proposed simulator uses *dl-load* technique, which allows dynamic loading of shared libraries in Linux. In Windows, this concept is represented by *dll* libraries. When a dynamic library is loaded at runtime, all its shared symbols can be used from within the main code. The *dl-load* technique is often used to develop application plugins which can be loaded in runtime.

In the proposed simulator, the only requirement to a user-developed *dl-library* is the presence of the *(modelname) create state(const char *statename)* method, where *(model-*

name) is the name of the model that is loaded. Each model name should be unique among other models used in the simulation to prevent namespace conflicts. The purpose of *(modelname) create state()* method is to create a state in runtime based on its name. This is the only convention that model developers have to follow in order to support dynamic model loading at runtime.

3.3 Model Context Dumps

It was previously mentioned that state variables of all models is stored in the *DataRepository* object, shared by all entities in the same simulation. Two important data handling conventions are adopted by the proposed simulator.

First, context dumps are implemented as simple dumps of memory occupied by model state variables to a file on the hard disk. Two files are created on each dump: model dump and data dump. Model dump is a simple structure which contains the name of the model, name of the model instance, and the name of the current state the model was blocking in at the time the dump was necessary. This is required to be able to load the model and recreate its condition adequately at some later time during simulation.

The second structure is a data dump, which is stored under a separate file consisting of a number of blocks each representing a state variable. The length of each variable type may be different, which is why each state variable is wrapped in a structure that contains its type in form of an integer value. Depending on the variable type, the appropriate number of bytes are dumped to of read from the hard disk. In the current version the simulator supports only *int*, *double*, *float*, and *string*. In future work some flexibility will be introduced including user-defined datatypes.

4. PERFORMANCE ANALYSIS

Time measurements were performed using *gettimeofday()* system call in Linux operating system. On the hardware used for the trials, the precision of this call was guaranteed to be better than $1ms$, which should be enough to see the difference in performance in the three stages of each simulation run above.

4.1 Simulation Topology and Setup

For performance verification, a very simple topology was used with 100K traffic generation nodes connected to a single sink node. Traffic was generated randomly in such a way that the combined throughput received by the sink would be around 1Gbps. Memory limit used to define when to unload models is set to 1MB, which is quite small given the number of nodes and, naturally, stimulates dynamic loading. This simple topology however allows for the analysis of load distribution by various parts of simulation cycles.

4.2 Distribution of Performance Load

Fig.4 contains aggregated statistics of the relative time share consumed by each action within each cycle. Since each simulation cycle consists of model load, execution of the user-defined code in the model, and model unload, only those three types of actions are compared. Naturally, loads and unloads happen only when the memory threshold of 1MB is exceeded. The histogram was created using all time measurements collected within a 30-second simulation.

At the first sight, the peaks in the histogram are around 10-30 percent for all three actions, but it is also interesting

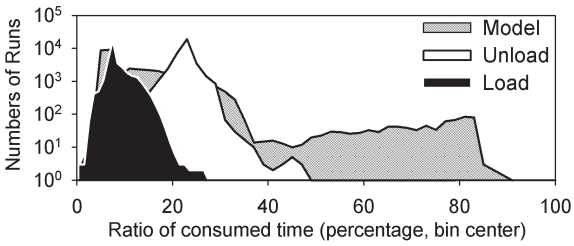

Figure 4: Histogram of time consumed by model loads and unloads, and work done within models.

that model loads and unloads are found only around their corresponding maximum points, while model interrupts are distributed all across the horizontal scale. In the right part of this distribution it always takes more time to execute the code of a model than to load and unload models.

Unloading in Fig.4 almost always takes more time due to the nature of each simulation loop. In each loop only one model may be loaded under the condition that the model is found to be offline at the moment. As for the unloading, if memory overflow was detected, several models may be unloaded until the memory falls back below its threshold. This alone makes unloading a more time consuming process.

5. CONCLUSIONS

The paper proposed a new simulator design which main feature is the ability to load and unload model instances at runtime. To be able to implement this in practice, two major changes to traditional simulator design had to be introduced.

A user-defined model in the proposed simulator is split into a number of *State* and *Condition* classes defined by user, and the basic *Model* class which is defined globally and statically by the simulator. In other words, the *Model* class plays the role of an interface between the simulator core and the user-defined code. This was practically implemented in form of defining the *State* and *Condition* classes as interfaces, which can be extended by states and transitions developed by users.

Another coding convention required for each user-developed model to define and implement a method which would return a *State* object based on the name of the state. Naturally, the first state in each model would be the *INIT* state, while all other states are allowed to have any name as long as it is uniquely defined within the user-defined model. This *State* factory code is to be implemented in form of a shared library to be dl-loaded by the simulator at runtime.

6. REFERENCES

[1] Omnet++. Available at: http://www.omnetpp.org/.

[2] L. B. Arief and N. A. Speirs. A UML tool for an automatic generation of simulation programs. In *Workshop on Software and Performance (WOSP)*, pages 71–76, Ottawa, Ontario, Canada, 2000.

[3] S. Balsamo and M. Marzolla. Simulation modeling of UML software architectures. In *European Simulation Multiconference*, Nottingham, UK, 2003.

[4] N. de Wet and P. Kritzinger. Using UML models for the performance analysis of network systems. In *Elsevier Computer Networks*, volume 49(5), pages 627–642, 2005.